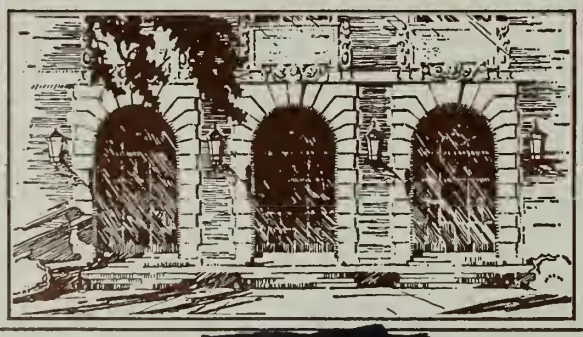
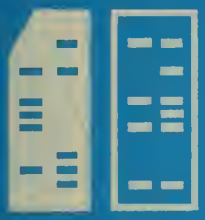


# A Revised ALGOL 68 Hardware Representation for ISO-code and EBCDIC

by

Wilfred J. Hansen

November, 1973





DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-73-607

# A Revised ALGOL 68 Hardware Representation for ISO-code and EBCDIC

by

Wilfred J. Hansen

November, 1973

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

This work was supported by the Department of Computer Science.

## Annotated Table of Contents

page

## Abstract

Because of the latitude allowed by the Revised ALGOL 68 Report, each implementation has a slightly different representation for the constructs of the language. This diversity can only lead to confuse as ALGOL 68 trained individuals find they need readaptation to programs at a new installation. The solution proposed here is to develop a single hardware representation which can be used on many computer systems. In fact this representation can conveniently be designed using only the intersection of the graphic characters available in the ISO code and EBCDIC.

The paper also proposes comfortable new representations for a few symbols and discusses the thorny problem of distinguishing bold face words.

# Preface

I didn't want to write this paper.

After the Los Angeles meeting of WG2.1 approved the Revised Report on the Algorithmic Language ALGOL 68 (1) I decided to spend two leisurely days designing a transportation representation for the language. It wasn't that easy (but I did it, it's another paper). Because the transportation representation is an encoding of ALGOL 68 program texts, an adequate explanation requires sample encoder and decoder programs. These must assume some specific hardware representation. Lacking a compiler I decided I could quickly design a suitable hardware presentation language of my own.

Older now, but wiser, I offer the following.

Sections three and four explain the hardware representation and sections one and two explain why I chose it. To a large extent, the sections can be read independently and the recommendations of one adopted without adopting any other. In particular, the transportation representation in no way depends on this hardware representation.

## Acknowledgments

This paper has been written with continuous reference to 'An ISO-Code Representation for ALGOL 68' by C. H. Lindsey (2). I am indebted to J. E. L. Peck for introducing me to ALGOL 68 and patiently explaining obscurities I encountered.

# I. Considerations in the Design of Hardware Representations

Now that the Revised Report has been approved by WG2.1, it is appropriate to reconsider the question of hardware representations of the language. (Hereafter, the Revised Report will be referred to as the Report; if there were any references to the earlier Report, they would specify the Original Report. Most remarks apply to both, anyway.) The Report has been written with the thought that it will be implemented on a wide variety of hardware with vastly differing character sets. As a consequence, it is not particularly specific as to how any construct will be represented on (say) cards.

Rather than envision the possibility of a tower of Babel of representations, I suggest that there is in fact a widely available set of graphics with which all constructs of the language can be represented. Selection of a widely available character set will bring these benefits:

° as trained ALGOL 68 programmers move from implementation to implementation, they will be able to begin programming and reading programs immediately and without confusion.

° as programs are sent from one installation to another, they will be understood without lengthy explanation and constant referral by the reader between the text and a codebook.

° many installations have a variety of devices with different character sets. Only by choice of a widely available representation will programmers be able to access the files containing their programs from all these devices, including both terminals and line printers.

° ALGOL 68 will more readily be accepted by outsiders as a single language and not a collection of similar languages.

It must be recognized that at the majority of installations, and especially in North America, ALGOL 68 will not be the primary language in use. For this reason it cannot be expected that the operating system will have provisions especially suited to the language. In particular, for many years ALGOL 68 by itself will not be a strong force determining the nature of character sets provided as standard by manufacturers.

The many considerations that affect design of a representation can be categorized into psychology, ALGOL 68, and hardware. These topics are covered in the remainder of section I. Section II suggests a few representations for symbols that may be controversial. It is important to note that the representation finally arrived at can be adjusted to take into account rejection of any of these suggestions. Section III details the decisions made in the rest of the representation and section IV discusses the sticky question of representations for bold-tags.

## I.1 Psychological considerations.

Care must be exercised in the design of representations for a number of general psychological reasons:

a) There are many possible sources of small confusions in representation design: odd characters, context dependent usage, dissimilar usage in similar contexts in similar languages, breaks in typing rhythms, and more. Each instance of confusion may be only a minor annoyance, or it may interrupt a train of thought and result in omission of critical phrases. Moreover, confusions may have a cummulative effect that can lead to frustration and breakdown in communication.

b) To some extent the physical representation and not the abstract 'strict language' is the medium of thought. This is true when writing a program and even more so when reading a program. Consequently, variations between physical representations ought to be carefully restricted.

c) When writing a program, a trained programmer writes by reflex and concentrates instead on the task at hand. For small changes of representation when changing installations, the retraining period is probably small, but if it can be avoided, everyone benefits.

d) Representations should be chosen with an eye to the representations used in other fields and in other languages. The task of attracting ALGOL 68 users is sufficiently difficult without repelling them with unusual symbol choices. (The Report has excellent symbol choices, I worry about implementations

Beyond these general considerations, the designer must keep in mind a number of human factors effects:

a) Some reasonably consistent aesthetic should be followed in the design to assist in readability. The aesthetic of the ALGOL 68 reference language is a pleasant combination of natural language and mathematical conventions. It seems characterized by economy of expression and avoidance of clutter.

b) Simultaneous with aesthetics, the designer must strive for understandability, unmistakability, and clarity.

c) A specific problem is that operators that bind closely ought to take less space than those that bind more loosely. In this regard, a bold tag used as an operator can suffer "stropping separation". This includes not only the length of the tag, but also the character(s) needed to strop it and any necessary blanks.

d) Another factor is the length of the text. Too short a text may correspond to a program that has been abbreviated beyond reason; but too long a text slows both the writer and the reader. A longer length for an infrequently used operator is acceptable because it will not substantially affect the length of the program.

e) No one is aided if an implementation provides too many alternative ways of expressing a single construct. Programmers are constantly forced to choose among alternatives; readers must be prepared to encounter that many more symbols. When text includes rare alternative form, the reader may remember it erroneously; he will at least have to interrupt his reading to try to recall the symbol.

Selection of specific symbols must depend on yet other factors:

a) The existence of groups of potential programmers trained in the meaning of a symbol. For example, '+' ought to mean addition because most potential users have studied algebra.

b) Use of the symbol in other widespread languages. Conflicting usage could be a source of confusion.

c) Relation of a symbol to its meaning ("graphic onomatopoeia"). For example, parentheses, braces, and brackets do appear to surround their contents.

d) The possibility of a confusing similarity between two graphics. Lindsey points out, for example, that '+' and '÷' appear alike on teletypes.

I.2. Decisions demanded by the Report.

In a number of areas, the Report leaves considerable latitude to accommodate implementations with varying character sets. The representation must specify what is allowed for other-string-item, STYLE-other-PRAGMENT-item, style-TALLY-letter-ABC, and style-TALLY-monad {R9.4a}. At least one means must be provided to write any operator in the standard prelude {R9.4b}, decisions must be made as to the values of abs, repr, 'null character', 'error char', 'flip', 'flop', 'blank', and 'max abs char' {R10.2.1}, and a conversion must be associated with 'stand conv' {10.3.1.2d}.

In 9.4 b, the Report accepts an implementation even if it provides only one of the alternatives for each symbol (say either '@' or at). The intention, in fact, seems to be to accept any representation language that provides at least one way of expressing each construct in the language. Thus an implementation need not necessarily have both stick-symbol and then-symbol, since with either one a choice-clause can be constructed {R9.1.1 c,h}. Similarly, a times-ten-to-the-power-symbol might be omitted since some letter-e-symbol will usually be available {R8.1.2.1 h}.

Selection of representations for standard prelude operators is complicated by the fact that not only do many operators have a number of symbols, but many symbols are assigned more than one function. Some way must be found through the mazes of relationships among uses and alternatives for '↑', '~', and '|'. Likewise, there are some complications if other than the reference representations are chosen for any of the multiple symbols that map into certain representations (for example, four symbols map into ':'). Some decision must be made as to whether the implementation will accept the "allowable" alternatives like '..' for ':'. {R9.4b}

Occasionally, a representation designer will be forced to consider use of a diphthong for some operation. In this effort, he must check the operator grammar in 9.4.2.1 of the Report to see that the diphthong is legal and thus will not cause ambiguity. A secondary consideration is to try to leave intact the possibility of families of operators. For example, diphthongs ending in '/' should be avoided because they are all available for a family:

+/ */ // -/ &/

Indeed, this is the family of APL reduction operators.

Finally, 9.4.2.2 b specifies that a bold-tag is composed of marks corresponding to its LETTER's and DIGIT's, where the 'mark corresponding to each LETTER ([or] DIGIT) is similar to the mark representing the corresponding LETTER-symbol ([or] DIGIT symbol)'. Interpretation of the word 'similar' has led to a variety of "stropping techniques". The representation designer must choose one of these techniques. My own suggestions will hinge on the observation that nothing is more similar to an object than itself, but see section IV for the gory details.

## I.3 Hardware Considerations.

Two standard codes for computer text have been defined and widely used: the ISO code (2) (and especially its ASCII subset (3,5)) and EBCDIC (4,5). The former are used by most of the world, and the latter is used by only one manufacturer. Tables defining these two codes are reproduced in figures 1, 2, and 3. Note that ISO has many national variants and the code provides spaces where national groups can place characters specific to their own needs.

An ALGOL 68 program will certainly not be given the same binary encoding in the two codes, since, for example, '+' is '00101110' in EBCDIC and '01010111' in ISO. However, in an important sense one can make programs in the two codes appear similar; the graphics chosen to represent each ALGOL 68 construct can be the same in both codes.

Examining the code tables, we see that ISO has a number of characters that will not be the same on every terminal, the so called "national characters". These interfere with a common graphic representation of ALGOL 68 so they should be avoided. Leaving them aside, the following characters are available in both codes:

upper and lower case letters: a-z A-Z

digits: 0-9

space ! " # $ % & ' ( ) * + , - . /

: ; < = > ? @ _

The following control characters appear in both codes and ought to be considered in design of a representation:

| | | | |
|---|---|---|---|
| BS | backspace | HT | horizontal tab |
| CR | carriage return | LF | linefeed |
| DEL | delete | NUL | null |
| FF | form feed | VT | vertical tab |

| | + 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 128 | 144 | 160 | 176 | 192 | 208 | 224 | 240 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | DS | | space | & | - | | | | | | | | | 0 |
| 1 | SOH | DC1 | SOS | | | | / | | a | j | | | A | J | | 1 |
| 2 | STX | DC2 | FS | SYN | | | | | b | k | s | | B | K | S | 2 |
| 3 | ETX | TM | | | | | | | c | l | t | | C | L | T | 3 |
| 4 | PF | RES | BYP | PN | | | | | d | m | u | | D | M | U | 4 |
| 5 | HT | NL | LF | RS | | | | | e | n | v | | E | N | V | 5 |
| 6 | LC | BS | ETB | UC | | | | | f | o | w | | F | O | W | 6 |
| 7 | DEL | IL | ESC | EOT | | | | | g | p | x | | G | P | X | 7 |
| 8 | | CAN | | | | | | | h | q | y | | H | Q | Y | 8 |
| 9 | | EM | | | | | | | i | r | z | | I | R | Z | 9 |
| 10 | SMM | CC | SM | | ¢ | ! | | : | | | | | | | | |
| 11 | VT | CU1 | CU2 | CU3 | . | $ | , | # | | | | | | | | |
| 12 | FF | IFS | | DC4 | < | * | % | @ | | | | | | | | |
| 13 | CR | IGS | ENQ | NAK | ( | ) | _ | ' | | | | | | | | |
| 14 | SO | IRS | ACK | | + | ; | > | = | | | | | | | | |
| 15 | SI | IUS | BEL | SUB | \| | ¬ | ? | " | | | | | | | | |

Figure 1. Extended Binary-Coded-Decimal Interchange Code (EBCDIC)

(Chart adapted from (4).)

| | 0+ | 16+ | 32+ | 48+ | 64+ | 80+ | 96+ | 112+ |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | space | 0 | (@) | P | (`) | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | (£) # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| 10 | LF | SUB | * | : | J | Z | j | z |
| 11 | VT | ESC | + | ; | K | ([) [ | k | { |
| 12 | FF | FS | , | < | L | \ | l | \| |
| 13 | CR | GS | - | = | M | (]) ] | m | } |
| 14 | SO | RS | . | > | N | (^) ^ | n | (¯) |
| 15 | SI | US | / | ? | O | _ | o | DEL |

Figure 2. ISO 7-bit Coded Character Set
(also known as ASCII)

Some codes are not officially assigned graphics, but the preferred alternatives are shown in parentheses. The right hand side of columns 32, 80, and 112 show the alternatives chosen for ASCII. (ISO chart taken from (2).)

| | | 35 | 64 | 91 | 92 | 93 | 94 | 96 | 123 | 124 | 125 | 126 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII | | # | @ | [ | \ | ] | ^ | ` | { | \| | } | ~ |
| Australia | | # | @ | [ | \ | ] | ^ | ` | { | \| | } | ~ |
| Denmark, Finland, Norway, Sweden | | | | Ä | Ö | Å | | | ä | ö | å | |
| France | 1 | £ | a | o | ç | § | ^ | ` | é | ù | é | ~ |
| | 2 | | | [ | \ | ] | | | | | | |
| W. Germany | 1 | # | @ | [ | \ | ] | ^ | ` | { | \| | } | – |
| | 2 | £ | § | Ä | Ö | Ü | | | ä | ö | ü | β |
| Japan | 1 | # | @ | [ | ¥ | ] | ^ | ` | { | \| | } | – |
| | 2 | £ | | | | | | # | | | | |
| New Zealand | | | @ | [ | | ] | ^ | ` | | | | – |
| United Kingdom | 1 | £ | @ | [ | \ | ] | ↑ | ` | { | \| | } | – |
| | 2 | | | | 10 | | | | | | | |
| | 3 | | | | ½ | | | | | | | |

Figure 3. ISO Alternatives Adopted by Various Countries

(Taken from Lindsey (2).)

A few comments on individual characters:

'#' is not ISO standard, but is the ASCII choice for position 35 in ISO. The alternative is '£', a currency symbol as is '¢'.

'@' is not ISO standard, but is the preferred alternative for position 64 and is only infrequently assigned other graphics.

'|', '!', '¬', and '~' Extended discussion of these follows.

The manufacturer supporting EBCDIC has helped introduce endless confusion into the codes for these graphics. The following charts illustrate this by giving the hexadecimal location in the code of various graphics.

| code: (references) | ISO (2) | ISO Alternatives (2) | ASCII-8 (3,5) | IBM ASCII-8 (4) | EBCDIC-8 (5) | IBM EBCDIC (4) |
|---|---|---|---|---|---|---|
| graphic | | | | | | |
| \| | 70 | ù ö | | 41 | | 4F |
| ! | 21 | | 21 | | 4F | |
| ¦ | | | 7C | FC | 6A | |
| ] | 5D | Ä Ü § | 5D | BD | 5A | |
| ¬ | | | | 6E | | 5F |
| ~ | 8E | – β | 8E | 8E | A1 | |
| ^ | 6E | ↑ | 6E | | 5F | |

Many terminal control software routines interpret the "best" meaning of characters. Thus on some North American timesharing systems a '!' at an ASCII terminal is converted to an EBCDIC '|' by the time it reaches the executing program. In such an environment or even in more benign environments there are then eight characters that can be produced by the system in response to some code that at one time was supposed to be '|': ! ¦ ] § Ä Ü ù ö. Similarly, '¬' can be ~ ^ – ↑ and β. This peregrination can render a character a poor choice for an important task in a representation language.

Another "feature" of terminals and terminal support routines is the translation of lower case characters to upper case because time-sharing supervisors expect upper case commands. This will influence the choice of letter symbols and stropping techniques below.

The final set of hardware problems is concerned with format effectors. In particular, should backspace and carriage return be permitted to instruct the compiler to reconstruct the input text image exactly as it would appear on a page produced by an ISO-code terminal? If so, bold stropping could be accomplished by backspacing and underlining, or even by returning the carriage and underlining. For a number of reasons, such composite characters are bothersome and must not be allowed:

° mechanically, backspace is a slow operation. For example an IBM Selectric® backspaces 42% slower than it forward spaces. This delay is enough to break typing rhythm and cause marginal discomfort and confusion.

° many terminals do not interpret ASCII backspace correctly and replace the character after repositioning the typing element (or cursor). R. W. Bemer has complained about this in a letter to Datamation (8), but I do not expect his plea to stem the tide.

° a compiler that interprets the printed image is not interpreting the characters in their sequential presentation. Error indications may not be correctly associated with the text, and even if so there may be no good clue as to how the text is stored in a file and should be modified. This problem would not be as bad if all editing were done interactively, but that is not always economical.

° some systems use only carriage return as a signal for end of line, so if the compiler were interpreting the image, all the characters would be on one line. (Certainly the compiler for such an installation would be more clever, but it is a curious thought.)

## II. Five ALGOL 68 Symbol Set Suggestions

### 1. plus-i-times-symbol: '+*'

Few devices provide '⊥', the reference language character for the plus-i-times-symbol. It can be constructed from overlaid characters, but these are a questionable recourse at best. The report also suggests i, but this suffers stropping separation or, if reserved words are used, conflicts with the identifier 'i'. The Lindsey ISO-code representation proposes '!' and '|_', but these encounter the ISO vertical bar uncertainty.

In Algol Bulletin 34 W. Freeman (6) proposed that modulo-operation be represented by '÷×'. This has been accepted in the Revised Report, and a parallel construction suggests '+×' for the plus-i-times-symbol. More usually, this will be written as '+*'; but note that this does not interfere with the x-asterisk family of diphthongs, because the family members '**' and '%*' already have standard meanings. Note: '+*' would also be used in transput to represent plus-i-times.

Examples: 3+*5 u +* v

(R 10.2.3.7j) (re a+re b) +* (im a+im b)

(R 11.1) (x ≥ 0 | rp+*ip | abs ip+*(y ≥ 0 | rp | -rp))

### 2. of-symbol: '-<'

At Los Angeles, '→' was removed as a representation of of-symbol because it points the wrong way - from son to parent, andbecause in practice it points the opposite way from the arrow in a similar construct in PL/I. Only of is left and it suffers extreme stropping separation for an operation that binds even more tightly than a monadic operator. One proposal for of-symbol is '.', but again this has the opposite meanings to the corresponding construct in PL/I. I derived an alternative to of by starting from the is-an-element-of-symbol, 'ϵ', and then considering '⋲' and '⋶'. The latter has an excellent diphthong: '-<', so I propose that the Report list of and '⋶' while the "approved alternative representation" be '-<'. Note: It should cause little difficulty that '⋶' is also the symbol for a photocathode.

Examples: father-<p

(R10.2.3.7c) sqrt(re-<a**2 + im-<a**2)

(R10.3.1.1d) p-<a > p-<b or p-<a = p-<b

and (1-<a > 1-<b

or 1-<a = 1-<b

and c-<a > c-<b)

3. stick-symbol, again-symbol, or-symbol

The Report proposes that the stick-symbol be represented by '|', a reasonable enough choice given the name. Three problems arise: the problems of '|' migration in ISO; the fact that PL/I uses this character for 'or', so it appears in boolean contexts in that language as well as in ALGOL 68; and the fact that the symbol has no inherent relation to decision. Despite these problems a brief-in-symbol is essential to avoid stropping separation in if-clauses yielding a value. The '?' meets all the objectives of the stick-symbol (which I would now rename the decision-symbol): it is available in most character sets without ambiguity, it is unused in other languages, and it naturally implies that a decision is being made.

A disadvantage of the stick-symbol is that it is also used for in and out in case-clauses, and '?' would be less meaningful in this context. Consider, however, the unfortunate similarity of '(a ? b, c ? d)' and '(a ? b; c ? d)'. (It is just as unfortunate with stick-symbols.) It would be too drastic to eliminate the brief case clauses, but why not specify that CASE-brief-in and CASE-brief-out be in and out? Then the example would be '(a in b, c out d)', a reasonable result, even with stropping separation. The case-again should not have a brief form: it occurs in long clauses that easily confuse a reader's parse; it is non-intuitive - an integral-again seemingly ought to select a new range for the original enquiry clause.

The proposal for choice clause tokens can be diagrammed:

| | start | in | again | out | finish |
|---|---|---|---|---|---|
| IF-brief | ( | ? | ?: | ? | ) |
| CASE-brief | ( | in | xxxxx | out | ) |
| IF-bold | if | then | elif | else | fi |
| CASE-bold | case | in | ouse | out | esac |

Reassigning the brief-in frees the stick-symbol for representation of or. For the time being, at least, I will not propose this because many systems will still use stick for decisions. I would, howevever, like to contradict a statement in the Lindsey ISO-code representation paper: "Hopefully, it [the or-symbol] is not as common as & [the and-symbol], so we shall restrict it to just or." In view of DeMorgan's laws, this assumption seems dubious. The proposed Revised Report (LA1(251) version 2) uses 47 and's and only 27 or's, but 14 of these and's are in checks to see whether a given operation can be performed on a given file. Changing these to or's is not only conceivable, but would enable the "good programming practice" of specifying the shortest alternative first. For example, 'put char' could begin:

```
if not opened of f or not put possible (f)

then undefined

else ...
```

Examples:

```
(R11.1)     (rp = 0 ? 0 ? y/(2 * rp))

(R10.3.1d)  (p of a < p of b ? false
                ?: p of a > p of b ? true
                ?: l of a < l of b ? false
                ?: l of a > l of b ? true
                ? c of a > c of b)

(R10.3.5.2 a string)
    (y(j) in
        (ref char c):
            (upb s = 1 ? c := s ? incomp := true) ,
        (ref () char cc):
            (upb s = upb cc - lwb cc ? cc (:) := s (:) ? incomp := true) ,
        (ref string ss): ss := s
        out incomp := true)
```

II.4. Disentangling '↓', '↑', '~', and '⌊'

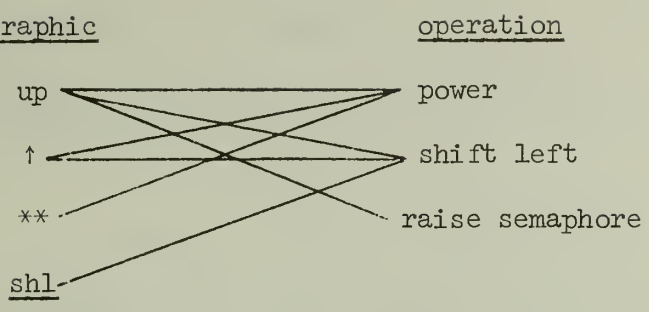
Because few devices provide these graphics, the Report allows a profusion of alternatives. The down-symbol, up-symbol, tilde-symbol, and floor-symbol are variously specified to replaceable by the bold symbols down, shr, up, shl, skip, not, lwb, and entier; also tilde may be synonymous with ¬ and up-arrow with **. Unfortunately, the synonymity is context dependent. For example entier may replace the floor-symbol in

```
⌊ (flex | al | xl) (1)
```

but the same replacement cannot be made in

```
⌊ (flex | al | xl).
```

The intertwining of symbols for exponentiation, shift left, and raise semaphore is even more complex. We can diagram the relations thus




An instance of up cannot readily be associated with the correct operation by either a reader or a compiler. Moreover, the hapless programmer is given little guidance as to how he might be able to write his program to avoid possible reader confusion.

These synonymities were introduced to permit an ALGOL 68 program to be written on any device, but only one representation is required at any one time. It is not difficult to assign symbols to operations so no symbol represents more than one operation. In addition to reducing reader ambiguity, this assignment simplifies construction of encoders for transportation representations. If the problems are resolved as follows, programs can still be represented on any device, but the synonymity conflicts are removed. Where possible, non-stropped representations are also proposed.

| | | |
|---|---|---|
| boolean negation | ¬ ~ not | as in the Report |
| skip | skip | no ~, the standard prelude does not use the tilde and has no problem |
| lower bound | lwb | no '⌊' |
| entier | entier | no '⌊' |

neither operation has a stronger claim to ⌊ than the other, and it would be silly to allow entier to replace lwb; this restriction will not make programs appreciably longer

| | | |
|---|---|---|
| lower semaphore | down | as in the Report |
| raise semaphore | up | as in the Report |
| shift left | shl | |
| shift right | shr | |

it is not intuitively clear why 'up' should mean 'left' and not 'right'; I considered suggesting '<*' and '*>', but their meanings could be forgotten since shifts are infrequent

| | | |
|---|---|---|
| exponentiation | ↑ ** pow | ** will seldom be unavailable; ↑ is kept because it looks right; pow will do if the others are unavailable. |

Examples:

```
(in an assembler)      op  shl 8 or adrs

(tautology test) (not (2 pow k /= 2rl shl k) | skip | undefined)

(binary search)

proc bisearch = (() int a, int arg, default) int:
     (int val := default; bits i := 2rl shl (bits width - 1);
     while abs i > upb a - lwb a + 1 do i := i shr 1 od;
     int k := abs i + lwb a - 1;
     while abs i /= 0 do i := i shr 1;
          (a(k) < arg | k := min (k+abs i, upb a)
          |: a(k) = arg | val := k;  i := bin 0 | k -:= abs i)
     od;  val)
```

II 5. 'flip': true, 'flop': false

For bool and bits, ALGOL 68 violates the principle that valid denotations should be valid transput values. As there seems to be no good reason for this, I propose the elimination of 'flip' and 'flop'. The value 'put' for true will be 'true'; that for false, 'false'. 'Get' for a bool will read a tag and set the bool to true if the tag starts with 't', false for 'f', and call 'char error mended' if the tag starts with neither. The transput value for a bits will be a BITS-denotation.

## III. The Design of the Hardware Representation

This section covers the points raised in section 4 of Lindsey's paper. The Representation is detailed in Appendix A, and section III.10 below contains some notes to explain that Appendix.

### III.1 Letter tokens

The central question here is whether to allow both upper and lower case letters in tags and the letters assigned denotation functions (a,b,c,d,e,f, g,i,k,l,n,p,q,r,s,x,y,z for radix, times ten to the power, and format markers). In fact few programmers will want to have tags differing only in the case of one or more letters. (Is there really enough difference between 'ox' and 'OX'? Is 'scan' not 'Scan'?) Moreover, many line printers are normally operated with only upper case. Therefore, the basic answer to the central question is that internally the compiler will have only one case of letter symbols (which one is immaterial). For string denotations and transput of strings, though, case will be distinguished. We now consider three categories of terminals and three contexts for tags and denotation function letters:

| | tag and denotation function letters | denotation function letters (a,b,c,d,e,f,r,t) input | output |
|---|---|---|---|
| UC/LC terminal, UC means bold | LC* | UC/LC* | LC |
| UC/LC terminal, case not used for stropping | UC/LC* | UC/LC* | LC |
| UC terminal | UC* | UC* | LC** |

*Converted to one case internally.
**System software or terminal converts to UC.

Appendix A lists lower case letters for ISO and upper case letters for EBCDIC. This is only because these characters sets are normally available on the corresponding equipment.

Of the national variant characters in the last four ISO columns, only '@' and '~' are assigned a function in this hardware representation. The other eight are available for use as additional letter-ABC-symbols. When they are not so used they can serve, as suggested below, as style-TALLY-monads and other-string-items.

III.2. Bold tags.

This hardware representation is designed to be applicable to any stropping convention. To facilitate this, only one case is assumed and no function is assigned to ''' and '_'. Stropping is discussed in section IV.

III.3. Composite characters.

Outlawed as discussed in section I.3.

III.4. Carriage Return, Line Feed, and Delete.

As Lindsey's paper suggests, CR, LF, CR/LF, LF/CR (my addition), form feed, and vertical tab all should terminate a line of input to the compiler. Delete, backspace, and all other control characters should be ignored.

III.5. Notes on Particular Representations

The controversial representations in this design have been discussed in section II. A few further choices deserve comment here.

'e' for times-ten-to-the-power-symbol

The Report implies that both 'e' and some other graphic ('•' or '\') will be available for this symbol. It does not seem valuable to have two or three graphics mean the same thing, especially where one is widely available and the others are not.

(space) for space-symbol

The Report proposes that both the space and visible space ('␣') be available. The latter cannot be allowed because it would have to be composite on available equipment. Using underlines for visible spaces does not help because they run together and are no easier to count than spaces.

'&' for and-symbol

This is reasonable and follows Lindsey's proposal.

lwb and entier for floor-symbol

These two bold tags are not interchangeable alternatives.

'¬' and '~' for not-symbol

Section I has described the ridiculous confusion as to the location of these symbols, including the possibilility that in some circumstances not-symbol would be typed as '↑'. One solution would be, as with stick-symbol, to suggest an alternate representation or to abolish the graphics in favor of not. The situation is not as serious with not-symbol, however, because

not-symbol is only an operator and its interpretation is not crucial to interpretation of the structure of the program text. Hence, '¬' and '~' are both allowed, depending on which character code is used. Note that, to avoid even further confusion, '~' is not a representation of the tilde-symbol and cannot be used in TAO's.

'?' for 'error char'

The report makes no proposal for this value. 'Error char' is used in transput to be the string value for a value that cannot be translated as specified. Is it not reasonable that this situation should be signalled with the ?

' ' for 'blank'

This assignment corresponds to the decision to use (space) for space-symbol.

'{' ... '}' for ISO brief-comment-symbol

The desirability of this representation is shown by PASCAL programs. Some brief-comment-symbol is necessary because '#' may not be available on all ISO terminals. There is no reason the compiler cannot insist that the left-brace start a comment and the right-brace end it.

III.6 Other-string-items, other-pragment-items.

The characters available for these will certainly vary depending on the code in use; I am not trying to define an implementation independent language. Essentially every character that can be typed should be allowed in these positions, except for the format effector characters, which only control the listing of the program. (Though a format effector could be a string value by means of transput or repr.) Except for the quote-image-symbol every character in a string will represent itself; there will be no composite characters or representations of one character by several.

III.7 Abs, repr, and conversion.

Abs and repr should certainly be designed so the position of a character in the code is its abs value. In ISO, abs "A" = 65 and repr 65 = "A"; in EBCDIC abs "A" = 193 and repr 193 = "A". Moreover, in both codes the null character will be repr 0. As far as possible, programs should be written so as to be independent of the actual values associated with abs and repr. As the encoder in my transportation representation paper shows, this can be achieved with minimal effort except for control codes. Appendix A proposes predefined identifiers for the six format affector control codes.

Lindsey's paper proposes two conversion regimes. 'Stand conv' transmits printable characters unchanged but ignores most control codes. The format codes would create the requisite spaces to position the text as it would appear on paper. Backspace would cause a call of 'char error' of the file. His other conversion was 'complete conv' which transmitted all characters

unchanged. Ignoring these, Lindsey's paper contains a program that uses what is called 'special conv'. In fact this latter is an important conversion that should be available for system programmers and anyone else who wants to know exactly how the user encoded spaces (and wants to avoid the processing time required to skip over them). I would call this conversion 'layout encoded conv': printable characters and format effectors are transmitted intact, but other control codes are ignored. When 'get'ing characters, each code is delivered in turn. When 'get'ing strings, a line is terminated by any format effector except horizontal tab; the terminating character is delivered as the last character of the line. If the line is terminated with CR/LF or LF/CR, both will appear at the end of the line. (This will depend on whether the operating system delivers both codes to the ALGOL 68 monitor.)

III.8. Use of remaining characters.

No virtue is attained by associating a standard meaning with every available character. Indeed, a few ought to remain available as style-TALLY-monads. The characters not yet assigned are

(ISO) [ ] ! | ∧ \

(EBCDIC) | ! ¢

Of these the brackets should not be monads because they would normally be used in pairs as delimiters, but the others can be monads.

It is unfortunate that the report makes no provision for style-TALLY-nomads because the given set (=,<,>,*,×,/) is limited and many of its combinations are in use. Indeed, of the characters above, it would not be unreasonable to specify that '!', '|', and '∧' could not be monadic, but could be nomads. With these new nomads there could be dyadic diphthongs like '*!', '-|', '/∧', and even '||'. Appendix A, however proposes no new nomads.

III.9. Representations with smaller character sets.

When only a subset of the characters assumed above is available, bold tags should be used to make up for missing characters.

| if this graphic is missing | these bold tags can replace its uses |
|---|---|
| % | over mod |
| ¬ ~ | not |
| @ | at |
| # | co |
| < | lt le of |
| > | gt ge |
| & | and |
| ? | then else elif |

If worst comes to worst, colon and semicolon will be missing and must be replaced with ugly diphthongs as suggested in R9.4b. Compilers should implement these only if numerous devices at the installation lack the specified graphic.

| the graphic | is replaced by |
|---|---|
| : | .. |
| ; | ., |
| := | .= |
| +=: | plusto |
| :=: | is |
| /=: | isnt |

(the second and third of these are no longer mentioned in the Report, but still do not lead to ambiguity. It can be remembered that semicolon is '.,' and not ',.' because real-denotations can start with '.' but cannot end with it, so that no ambiguity exists in '3.,5' for '3;5'.)

In cases where standard graphics are absent but other symbols are available, the temptation to use the latter should be resisted with passion.

The minimum character set for ALGOL 68 is thus:

letters, digits, =, +, -, *, ,, ., (, ), $, /, ", and possibly some stropping character.

III.10. Guide to reading Appendix A.

Some symbols in Appendix A are marked boldface by underlining them. They are to be keywords, reserved words, or stropped words, according to whatever convention is adopted.

The first listed "representation" for a symbol should be used if possible. If no representation is possible (and the symbol is not intended as a MONAD), the first possible graphic listed in the "alternatives" column should be chosen. Up-symbol and down-symbol have neither representation nor alternative and cannot be written in a program {see section II.4}.

Please note that "typographic display features" (spaces, new lines, and new pages) cannot divide the characters of a diphthong or bold tag.

Numerals in braces refer to sections of this paper where decisions are discussed. Where no comment is made, the representation follows the suggestions of the Revised Report.

## IV. Stropping Recommendations

ALGOL 60 invented the concept of indivisible symbols represented by boldface identifiers. ALGOL 68 extended this concept to allow the user to invent his own bold tags. In the Revised Report, it is made clear that these symbols are no longer indivisible, but are composed of ordinary letters possibly set off in some way to indicate their type face. The Report does not suggest how they are to be set off, but gives five examples {R9.4.2.2b} including one using script letters, a feature found on few existing computer transput devices.

Implementations have adopted diverse stropping conventions; some have even attempted to implement several conventions with provision for pragmatic selection of one at compile time. Such attempts at universality seem doomed to failure, however. Without much effort I was able to write a list of twenty different stropping conventions, most incompatible with two-thirds of the rest. Instead, the compiler writer should implement one convention. Moreover, eachprogram will at one time or another be transput on every device in the installation, so the stropping convention chosen must be applicable to every device.

Traditionally, ALGOL 60 programs have been stropped with apostrophes at the beginning and end of each boldface symbol. To me this convention seems unduly cluttering. I have always been struck by the clean aesthetics of ALGOL: short symbols, indentation emphasized as a tool, no semi-colon before else, ALGOL 68's brief comment forms, the top-down structure of program texts. In this atmosphere, apostrophes seem as appropriate as neon lights in a Japanese garden. Examine

```
'begin' 'real' x; 'char' c;

   get (f, (c,x));

   'if' c = "i" 'then' x+1 'else' x 'fi' 'end'
```

Note that in the natural (Western) way of looking top-to-bottom and left-to-right, the first item the eye encounters is not a piece of information, but is the interspersive apostrophe. Indeed, the most important distinction between pairs of words is their first letters, but the first 'letter' of all apostrophied words is the same. Note also that a hurried programmer who used other languages might waste time over the confusion between 'if' and "i"; not much time, but perhaps enough to lose his train of thought. Moreover, there is no one apostrophe stropping method. Lindsey lists three, and my list included five, only one of which has been made illegal by the Revised Report.

Another stropping technique that has been proposed is underlining; it does not clutter and has a rather pleasant appearance. Its difficulties are not aesthetic, but operational: it takes longer to enter and revise a program with underline stropping, many text editors and line printer routines do not support underlining. Mechanical backspacing is a slow operation. Finally, underlining constructs composite characters and suffers the disadvantages of listings that are not in one-one correspondence with the file.

Apostrophes clutter. Underlining has operational problems. I reject them both for stropping. (Blithely ignoring the derivation of the word 'stropping'.) What do I propose instead? Reserved words. However, many implementers will feel they must provide some stropping convention so I discuss below four conventions in order of increasing preference.

IV.1. Postfix underline is bold (least favored, but better than sixteen other schemes).

The least intrusive stropping scheme is to append an underline to the end of a bold word. The reader can easily ignore the character, but it is there to resolve ambiguity if he needs it. A trained ALGOL reader scans the indentation before examining the text. With prefix stropping, most lines begin with the low-information-content strop character; with postfix stropping, the strop character is out of the way.

Our earlier random program would look like this:

```
begin_real_x;  char_c;

    get (f, (x, c));

    if_ c="i" then_ x+1 else_ x fi_ end_
```

Note that '_' alone is enough and no space is needed. One of the advantages of this notation to the programmer-cum-keypuncher is that the underline is typed in the same sequence that it would be drawn: last after the rest of the word. Thus written work could still underline bold words and the transliteration while keypunching is not onerous.

In a small way, this convention is compatible with complete underlining of bold words. If backspaces are ignored and multiple underlines converted to a single underline, the compiler can accept "begin ββββββ _____" (where 'β' means backspace) as equivalent to "begin_". But the user could not type "bβ_eβ_gβ_iβ_nβ_" and achieve the same effect. (Alternatively, the latter could mean begin and a non-letter be required to end a bold tag. This introduces undue stropping separation in, for example, "re of_ z".)

Postfix underline stropping is poor for terminals with a non-escaping underline key. They would have to type "begin_ " which would be printed as "begin_". Presumably implementors for such terminals would choose some convention later in this section.

IV.2. Upper case is bold.

One convention gaining wide use in Europe is case stropping: bold tags in upper case and tags in lower case.

```
BEGIN REAL x;  CHAR c;

    get (f, (c, x));

    IF c="i" THEN x+1 ELSE x FI END
```

If the tags are chosen to be real words, they look more like normal text if they are lower case. (Theoretically, no space would be required between "REAL" and "x", but it might be considered gauche to omit it. Consider "reOFz".)

Case stropping is excellent, except that existing devices, especially line printers, often support only upper case. One could postulate combining this technique with underlining or apostrophes to be used on an upper case only device. Unfortunately, this fails because in systems with only one case, it is usually upper case; plain tags would be upper case and thus mistaken for bold.

IV.3. Lower case or postfix underline is bold.

A reversal of the previous convention allows combination of both the first two conventions. Bold words would be lower case (as they are in the Report) or would be followed by an underline:

```
BEGIN_real X; char C;
    GET (F, (X,C));
    if C="i" THEN_ X+1 else X fi end
```

(The mixed case if-THEN_-else-fi looks unpleasant and is. I postulate that in this particular case it was forced on the programmer because he was fixing the "THEN_" at an upper case terminal.) Presumably, the user would use case distinction at a mixed case terminal and fall back on postfix underline at an upper-case-only terminal. An upper case line printer would print all as upper case, and the user would have to rely on context to distinguish bold from roman. The latter is never an arduous chore in a well written program.

The only disadvantage with this third stropping convention is that the second, contradictory, convention has achieved considerable useage.

IV.4. Reserved words are bold (most preferred technique).

Few languages other than ALGOL have stropping conventions, most rely on reserved words or a language design that makes all distinctions determinable from immediate context. Many successful ALGOL compilers have avoided stropping. Several ALGOL 68 implementations (at least Vancouver, Dartmouth, Illinois Institute of Technology, and probably many more) have worked out techniques to minimize or eliminate stropping. Here is a random program without stropping:

```
begin real x; char c;
    get (f, (x, c));
    if c = "i" then x+1 else x fi end
```

Is there any reason to believe that the human reader should have difficulty distinguishing bold from roman in this program?

One interesting point is the fact that the Revised Report lists 20% more predefined roman words than bold tags. To be sure, the roman words can all be redefined with far less penalty than a redefinition of if; but still they constitute a large class of identifiers the user must remember.

In the past, automatically generated parsers have behaved poorly when confronted with a reserved word used as an identifier. There is no inherent need for this to be the case, and work such as that reported by Graham and Rhodes (7) is showing how to avoid these problems.

What is to be done about identifiers that contain a reserved word as an integral part, for example "year to date"? It is possible to forbid this, but that leaves too many opportunities for the programmer to forget that something is a reserved word. After all, the entire identifier is far from any reserved word. Instead, I suggest that underlines replace blanks in identifiers. Where an identifier is continued from one line to the next, it may have embedded blanks, but it must end with an underline on the first line or begin with an underline on the second line. Either

```
year_to_
    date
```

or

```
year_to
_date
```

will be acceptable renderings of "year_to_date".

In ALGOL 68 a user can declare arbitrary tags to be bold face. There are three reasonable techniques for dealing with these. (1) Such tags would be unstropped and treated as bold only in the block where they were so declared. This technique implies that the token scanner will analyze the block structure, a good idea because it can aid automatic correction of bracket errors. (2) They could be unstropped, but treated as bold throughout the compilation. At least one implementation (IIT) has chosen this route. (3) They could be stropped in some way. Underlines serve as spaces in roman tags, so they cannot be used; apostrophes at both ends are unnecessary; postfix apostrophes are almost as unobtrusive as postfix underlines; so I propose that a postfix apostrophe be used.

Much as I dislike stropping of standard bold tags, I recommend the third method of distinguishing user created bold tags. They will certainly be less used than syntactic words like if and real, and usually they will not appear more than once in a phrase. More critically, they will be unfamiliar to a reader of the program, so he deserves to have them set off in some manner. Here is a part of a program:

```
mode token' = struct (int type, string val, rtok'next),
     rtok' = ref token';
rtok'toklist := nil, eop := heap token';
ref rtok'tokput := toklist;
sema tokens_ready = level 0,
     may_move_tokens = level 1;
```

Note that in its use as a mode or a monadic operator, a user defined bold tag is separated from its object by the apostrophe. It can be viewed as a notation that the thing to its left operates on the thing to its right. Moreover, a postfix apostrophe is somewhat like normal usage where an apostrophe may appear near the end of a word.

Some emergency method of stropping may be necessary anyway to handle cases where a tag is declared both bold and roman. In such cases, it would be assumed roman unless stropped or unless the syntactic position demanded that it be bold. This mechanism is one way to solve the problem of 're' and re and 'im' and im.

(Writing this, I have come to worry about "re" and friends. Why do both operator and field selector exist? Synonymously, is compl a primitive mode or is it a struct? If it is a mode then we need only operators to process it and an operator to construct objects of that modes (which we have: '|'). Viewed thus, it makes no sense to assign a value to part of the primitive object "z". On the other hand, given the stropping separation of re and im and given "-<" for of, it is just as reasonable to eliminate re and im from the language.)

Final thought: I believe an efficient compiler can be written so that no more than 29 words are reserved:

BEGIN, BY, CASE, CO, COMMENT, DØ, ELIF, ELSE, END, ESAC, FI,
FLEX, FOR, FROM, IF, IN, MODE, OD, OP, OUSE, OUT, PR, PRAGMAT,
PRIO, PROC, REF, STRUCT, THEN, TO, UNION, WHILE.

With these words, the structure of the text can be determined. The other bold words in the Report could be redeclared, but could not be used in their bold sense within that block.

I offer this limited-reserved-words approach as a challenge to parser implementers.

## References

(1) van Wijngaarden, A., et al., Almost the Revised Report on the Algorithmic Lanugage ALGOL 68, private communication, (Sept., 1973). This version is slightly more recent than the version considered at Los Angeles and includes most of the corrections agreed on there.

(2) Lindsey, C. H., 'An ISO-Code Representation for ALGOL 68', ALGOL Bulletin 31 (March, 1970), pp. 37-60.

(3) ANSI, 'Data Communication Control Procedures for the USA Standard Code for Information Interchange', CACM 12, 3 (March, 1969), pp. 166-178. ASCII is listed in Appendix-E.

(4) IBM Corp., IBM System/360 Principles of Operation, Order No. GA22-6821-8, 1970, pp. 150.2-150.3. (Note that the graphic '.' has been omitted from position '4B' of the document.)

(5) ANSI, 'Correspondences of 8-Bit and Hollerith Codes for Computer Environments - A USASI Tutorial', CACM 11, 11 (Nov., 1968), pp. 783-789. Corrected in CACM 12, 5 (May, 1969), p. 294.

(6) Freeman, W., 'Suggestions regarding certain representations in ALGOL 68', ALGOL Bulletin 34 (July, 1972), pp. 41-44.

(7) Graham, S. L. and S. P. Rhodes, 'Practical Syntactic Error Recovery in Compilers', Conference Record of ACM Symposium on Principles of Programming Languages, Boston, Massachusetts (Oct., 1973), pp. 52-58.

(8) Bemer, R. W., 'Backspace Bungle', Datamation 19, 9 (September, 1973), p. 25.

## Appendix A. Proposed Hardware Representation

Where there is a difference between the ISO and EBCDIC representations, the respective parts are labeled (ISO) and (EBCDIC).

8.1.4 Character denotations {III.6}

d) other string item : style i monad {942b};

& / = < > % * : ; ? @ $ # _ '
(ISO) ~ [ ] { } (and upper case letters)
(EBCDIC) ¬ (non-printing codes above 'space'; lower case letters).

9.1.1 Syntax {III.3}

A) IF :: choice using boolean.
a) CHOICE brief start : open token.
b) IF bold start : if token.
c) CASE bold start : case token.
d) IF brief in : decision token.
e) IF bold in : then token.
f) CASE STYLE in : in token.
g) IF brief again : again token.
h) IF bold again : else if token.
i) CASE bold again : out case token.
j) IF brief out : decision token.
k) IF bold out : else token.
l) CASE STYLE out : out token.
m) CHOICE brief finish : close token.
n) IF bold finish : fi token.
o) CASE bold finish : esac token.

{there is no 'CASE brief again'. The provisions above can be diagrammed:

| | start | in | again | out | finish |
|---|---|---|---|---|---|
| IF brief | ( | ? | ?: | ? | ) |
| CASE brief | ( | in | | out | ) |
| IF bold | if | then | elif | else | fi |
| CASE bold | case | in | ouse | out | esac |

}

9.2 Comments and pragmats {III.6}

d) STYLE other PRAGMENT item : quote symbol;

other string item (except not STYLE-PRAGMENT-symbol).

9.4.2 Other TAX symbols

a) style i letter ABC : {if case is not used for stropping} {III.1}
(ISO) upper case of letter ABC symbol
(EBCDIC) lower case of letter ABC symbol
{internally all STYLE-letter-ABC's are converted to one case}.

b) style i monad : (ISO) ! \ | ^ {III.8}

(EBCDIC) | ! ¢ .

9.4.2.2 Representation {IV}

b) Stropping. {This representation is compatible with all proposed stropping conventions and favors none (pun intended).}

9.4.1 Representations of symbols

a) Letter symbols {III.1}

| symbol | (ISO) | (EBCDIC) | symbol | (ISO) | (EBCDIC) |
|---|---|---|---|---|---|
| letter a symbol | a | A | letter n symbol | n | N |
| letter b symbol | b | B | letter o symbol | o | O |
| letter c symbol | c | C | letter p symbol | p | P |
| letter d symbol | d | D | letter q symbol | q | Q |
| letter e symbol | e | E | letter r symbol | r | R |
| letter f symbol | f | F | letter s symbol | s | S |
| letter g symbol | g | G | letter t symbol | t | T |
| letter h symbol | h | H | letter u symbol | u | U |
| letter i symbol | i | I | letter v symbol | v | V |
| letter j symbol | j | J | letter w symbol | w | W |
| letter k symbol | k | K | letter x symbol | x | X |
| letter l symbol | l | L | letter y symbol | y | Y |
| letter m symbol | m | M | letter z symbol | z | Z |

b) Denotation symbols

| symbol | representation |
|---|---|
| digit zero symbol | 0 |
| digit one symbol | 1 |
| digit two symbol | 2 |
| digit three symbol | 3 |
| digit four symbol | 4 |
| digit five symbol | 5 |
| digit six symbol | 6 |
| digit seven symbol | 7 |
| digit eight symbol | 8 |
| digit nine symbol | 9 |
| point symbol | . |
| times ten to the power symbol | (ISO) e (EBCDIC) E {III.5} |

| symbol | representation | | |
|---|---|---|---|
| true symbol | true | | |
| false symbol | false | | |
| quote symbol | " | | |
| quote image symbol | "" | | |
| space symbol | (space) | | {III.5} |
| comma symbol | , | | |
| empty symbol | empty | | |

c) Operator symbols

| symbol | representation | alternates | |
|---|---|---|---|
| or symbol | | or | {III.5} |
| and symbol | | & and | |
| ampersand symbol | & | | |
| differs from symbol | | /= ne | |
| is less than symbol | < | lt | |
| is at most symbol | | <= le | |
| is at least symbol | | >= ge | |
| is greater than symbol | > | gt | |
| divided by symbol | / | | |
| over symbol | | % over | |
| percent symbol | % | | |
| window symbol | | elem | |
| floor symbol | | lwb, entier | {II.4, III.5} |
| ceiling symbol | | upb | {II.4} |
| plus i times symbol | | +* | {II.1} |
| not symbol | (EBCDIC) ¬ | (ISO) ~ not | {II.4, III.5} |
| tilde symbol | | | |
| down symbol | | | {II.4} |
| up symbol | | | |
| plus symbol | + | | |
| minus symbol | - | | |
| equals symbol | = | eq | |
| times symbol | | * | |
| asterisk symbol | * | | |
| assigns to symbol | =: | | |
| becomes symbol | := | | |

{Although they are not listed in 9.4, the following are defined in chapter 10.}

| | | | |
|---|---|---|---|
| exponentiation operator | ** | pow ↑ | {II.4} |
| modulo operator | %* | mod | |
| plus and becomes operator | +:= | plusab | |
| minus and becomes operator | -:= | minusab | |
| times and becomes operator | *:= | timesab | |
| divided by and becomes operator | /:= | divab | |
| over and becomes operator | %:= | overab | |
| modulo and becomes operator | %*:= | modab | |
| plus to operator | +=: | plus to | |
| shift left operator | | shl | {II.4} |
| shift right operator | | shl | |
| raise semaphore operator | | up | |
| lower semaphore operator | | down | |

d) Declaration symbols

As in the Revised Report

e) Mode standards

As in the Revised Report

f) Syntactic symbols

| symbol | representation | | |
|---|---|---|---|
| bold begin symbol | begin | | |
| bold end symbol | end | | |
| brief begin symbol | ( | | |
| brief end symbol | ) | | |
| and also symbol | , | | |
| goon symbol | ; | | |
| completion symbol | exit | | |
| label symbol | : | | |
| parallel symbol | par | | |
| open symbol | ( | | |
| close symbol | ) | | |
| decision symbol | ? | | } {II.3} |
| again symbol | ?: | | |
| if symbol | if | | |
| then symbol | then | | |
| else if symbol | elif | | |
| else symbol | else | | |
| fi symbol | fi | | |
| case symbol | case | | |
| in symbol | in | | |
| out case symbol | ouse | | |
| out symbol | out | | |
| esac symbol | esac | | |
| colon symbol | : | | |
| brief sub symbol | | | |
| brief bus symbol | | | |
| style i sub symbol | ( | | |
| style i bus symbol | ) | | |
| up to symbol | : | | |
| at symbol | @ | at | |
| is symbol | :=: | is | |
| is not symbol | :/=: | isnt | |
| nil symbol | nil | | |
| of symbol | ≺ | of | {II.3} |
| routine symbol | : | | |
| go to symbol | goto | | |
| go symbol | go | | |
| skip symbol | skip | | |
| formatter symbol | $ | | |

g) Loop symbols
No change from Report

h) Pragment symbols

| symbol | representation |
|---|---|
| brief comment symbol | (ISO) {...} |
| bold comment symbol | comment |
| style i comment symbol | co |
| style ii comment symbol | # |
| bold pragmat symbol | pragmat |
| style i pragmat symbol | pr |

{III.5}

10.2.1 Environment enquiries

```
p) int max abs char = (ISO) 127  (EBCDIC) 255;
q) char null character = repr 0;
r) char flip = "t";
s) char flop = "f";
t) char errorchar = "?";
u) char blank = " ";
v) char horizontal tab = repr ((EBCDIC) 5   (ISO) 9),
       backspace = repr ((EBCDIC) 22   (ISO) 8),
       carriage return = repr 13,
       line feed = repr ((EBCDIC) 37   (ISO) 10),
       vertical tab = repr 11,
       form feed = repr 12;
```

10.6.1. Library Preludes

```
a) proc complete conv = (ref book b) conv:
       (conv c;
       for i from 0 to max abs char do
           (aleph of c) (i) := (repr i, repr i) od;
       c);

b) proc layout encoded conv = (ref book b) conv:
       # characters to be ignored are set to null #
       (conv c;
       for i from 0 to abs blank - 1 do
           (aleph of c) (i) := (null character, repr i) od;
       for i to 6 do
           char ch = (i in horizontal tab, backspace,
               carriage return, line feed, vertical tab,
               form feed);
           (aleph of c) (abs ch) := (c, c) od;
       for i from abs blank to max abs char do
           (aleph of c) (i) := (repr i, repr i) od;
       c);
```

| OGRAPHIC DATA T | 1. Report No. UIUCDCS-R-73-607 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|
| e and Subtitle | | | 5. Report Date November, 1973 |
| A Revised ALGOL 68 Hardware Representation for ISO-code and EBDCID | | | 6. |
| nor(s) Wilfred J. Hansen | | | 8. Performing Organization Rept. No. |
| orming Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois | | | 10. Project/Task/Work Unit No. |
| | | | 11. Contract/Grant No. |
| nsoring Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois | | | 13. Type of Report & Period Covered |
| | | | 14. |

plementary Notes

stracts

Because of the latitude allowed by the Revised ALGOL 68 Report, each lementation has a slightly different representation for the constructs of the uage. This diversity can only lead to confusion as ALGOL 68 trained individuals they need readaptation to program at a new installation. The solution osed here is to develop a single hardware representation which can be used any computer systems. In fact this representation can conveniently be designed g only the intersection of the graphic characters available in the ISO code EBCDIC.

The paper also proposes comfortable new representations for a few symbols discusses the thorny problem of distinguishing bold face words.

y Words and Document Analysis. 17a. Descriptors

ALGOL68,
hardware representation, program interchange,
symbols, characters,
bold face letters,
ASCII, ISO-code, EBCDIC

entifiers/Open-Ended Terms

OSATI Field/Group

| ilability Statement | 19. Security Class (This Report) UNCLASSIFIED | 21. No. of Pages |
|---|---|---|
| | 20. Security Class (This Page UNCLASSIFIED | 22. Price |

TIS-35 (10-70) USCOMM-DC 40329-P71